LEARN C++
Skill up with our free tutorials

2.5 — Introduction to local scope

ALEX[1] APRIL 8, 2024

Local variables

Variables defined inside the body of a function are called **local variables** (as opposed to *global variables*, which we'll discuss in a future chapter):

```cpp
int add(int x, int y)
{
    int z{ x + y }; // z is a local variable

    return z;
}
```

Function parameters are also generally considered to be local variables, and we will include them as such:

```cpp
int add(int x, int y) // function parameters x and y are local variables
{
    int z{ x + y };

    return z;
}
```

In this lesson, we'll take a look at some properties of local variables in more detail.

Local variable lifetime

In lesson 1.3 -- Introduction to objects and variables (https://www.learncpp.com/cpp-tutorial/introduction-to-objects-and-variables/)[2], we discussed how a variable definition such as `int x;` causes the variable to be instantiated (created) when this statement is executed. Function parameters are created and initialized when the function is entered, and variables within the function body are created and initialized at the point of definition.

For example:

```cpp
int add(int x, int y) // x and y created and initialized here
{
    int z{ x + y };   // z created and initialized here

    return z;
}
```

The natural follow-up question is, “so when is an instantiated variable destroyed?”. Local variables are destroyed in the opposite order of creation at the end of the set of curly braces in which it is defined (or for a function parameter, at the end of the function).

```
int add(int x, int y)
{
    int z{ x + y };

    return z;
} // z, y, and x destroyed here
```

Much like a person’s lifetime is defined to be the time between their birth and death, an object’s **lifetime** is defined to be the time between its creation and destruction. Note that variable creation and destruction happen when the program is running (called runtime), not at compile time. Therefore, lifetime is a runtime property.

For advanced readers

The above rules around creation, initialization, and destruction are guarantees. That is, objects must be created and initialized no later than the point of definition, and destroyed no earlier than the end of the set of the curly braces in which they are defined (or, for function parameters, at the end of the function).

In actuality, the C++ specification gives compilers a lot of flexibility to determine when local variables are created and destroyed. Objects may be created earlier, or destroyed later for optimization purposes. Most often, local variables are created when the function is entered, and destroyed in the opposite order of creation when the function is exited. We’ll discuss this in more detail in a future lesson, when we talk about the call stack.

Here’s a slightly more complex program demonstrating the lifetime of a variable named `x`:

```
#include <iostream>

void doSomething()
{
    std::cout << "Hello!\n";
}

int main()
{
    int x{ 0 };    // x's lifetime begins here

    doSomething(); // x is still alive during this function call

    return 0;
} // x's lifetime ends here
```

In the above program, the lifetime of `x` runs from the point of definition to the end of function `main`. This includes the time spent during the execution of function `doSomething`.

What happens when an object is destroyed?

In most cases, nothing. The destroyed object becomes invalid, and any further use of the object will result in undefined behavior. At some point after destruction, the memory used by the object will be freed up for reuse.

For advanced readers

If the object is a class type object, prior to destruction, a special function called a destructor is invoked. In many cases, the destructor does nothing, in which case no cost is incurred.

Local scope

An identifier's **scope** determines where the identifier can be seen and used within the source code. When an identifier can be seen and used, we say it is **in scope**. When an identifier can not be seen, we can not use it, and we say it is **out of scope**. Scope is a compile-time property, and trying to use an identifier when it is not in scope will result in a compile error.

A local variable's scope begins at the point of variable definition, and stops at the end of the set of curly braces in which it is defined (or for function parameters, at the end of the function). This ensures variables can not be used before the point of definition (even if the compiler opts to create them before then). Local variables defined in one function are also not in scope in other functions that are called.

Here's a program demonstrating the scope of a variable named `x`:

```cpp
#include <iostream>

// x is not in scope anywhere in this function
void doSomething()
{
    std::cout << "Hello!\n";
}

int main()
{
    // x can not be used here because it's not in scope yet

    int x{ 0 }; // x enters scope here and can now be used within this
function

    doSomething();

    return 0;
} // x goes out of scope here and can no longer be used
```

In the above program, variable `x` enters scope at the point of definition and goes out of scope at the end of the `main` function. Note that variable `x` is not in scope anywhere inside of function `doSomething`. The fact that function `main` calls function `doSomething` is irrelevant in this context.

"Out of scope" vs "going out of scope"

The terms "out of scope" and "going out of scope" can be confusing to new programmers.

An identifier is out of scope anywhere it cannot be accessed within the code. In the example above, the identifier `x` is in scope from its point of definition to the end of the `main` function. The identifier `x` is out of scope outside of that code region.

The term "going out of scope" is typically applied to objects rather than identifiers. We say an object goes out of scope at the end of the scope (the end curly brace) in which the object was instantiated. In the example above, the object named `x` goes out of scope at the end of the function `main`.

A local variable's lifetime ends at the point where it goes out of scope, so local variables are destroyed at this point.

Note that not all types of variables are destroyed when they go out of scope. We'll see examples of these in future lessons.

Another example

Here's a slightly more complex example. Remember, lifetime is a runtime property, and scope is a compile-time property, so although we are talking about both in the same program, they are enforced at different points.

```cpp
#include <iostream>

int add(int x, int y) // x and y are created and enter scope here
{
    // x and y are visible/usable within this function only
    return x + y;
} // y and x go out of scope and are destroyed here

int main()
{
    int a{ 5 }; // a is created, initialized, and enters scope here
    int b{ 6 }; // b is created, initialized, and enters scope here

    // a and b are usable within this function only
    std::cout << add(a, b) << '\n'; // calls function add() with x=5 and y=6

    return 0;
} // b and a go out of scope and are destroyed here
```

Parameters `x` and `y` are created when the `add` function is called, can only be seen/used within function `add`, and are destroyed at the end of `add`. Variables `a` and `b` are created within function `main`, can only be seen/used within function `main`, and are destroyed at the end of `main`.

To enhance your understanding of how all this fits together, let's trace through this program in a little more detail. The following happens, in order:

- Execution starts at the top of `main`.
- `main` variable `a` is created and given value `5`.
- `main` variable `b` is created and given value `6`.
- Function `add` is called with argument values `5` and `6`.

- `add` parameters `x` and `y` are created and initialized with values `5` and `6` respectively.
- The expression `x + y` is evaluated to produce the value `11`.
- `add` copies the value `11` back to caller `main`.
- `add` parameters `y` and `x` are destroyed.
- `main` prints `11` to the console.
- `main` returns `0` to the operating system.
- `main` variables `b` and `a` are destroyed.

And we're done.

Note that if function `add` were to be called twice, parameters `x` and `y` would be created and destroyed twice -- once for each call. In a program with lots of functions and function calls, variables are created and destroyed often.

Functional separation

In the above example, it's easy to see that variables `a` and `b` are different variables from `x` and `y`.

Now consider the following similar program:

```cpp
#include <iostream>

int add(int x, int y) // add's x and y are created and enter scope here
{
    // add's x and y are visible/usable within this function only
    return x + y;
} // add's y and x go out of scope and are destroyed here

int main()
{
    int x{ 5 }; // main's x is created, initialized, and enters scope here
    int y{ 6 }; // main's y is created, initialized, and enters scope here

    // main's x and y are usable within this function only
    std::cout << add(x, y) << '\n'; // calls function add() with x=5 and y=6

    return 0;
} // main's y and x go out of scope and are destroyed here
```

In this example, all we've done is change the names of variables `a` and `b` inside of function `main` to `x` and `y`. This program compiles and runs identically, even though functions `main` and `add` both have variables named `x` and `y`. Why does this work?

First, we need to recognize that even though functions `main` and `add` both have variables named `x` and `y`, these variables are distinct. The `x` and `y` in function `main` have nothing to do with the `x` and `y` in function `add` -- they just happen to share the same names.

Second, when inside of function `main`, the names `x` and `y` refer to main's locally scoped variables `x` and `y`. Those variables can only be seen (and used) inside of `main`. Similarly, when inside function `add`, the names `x` and `y` refer to function parameters `x` and `y`, which can only be seen (and used) inside of `add`.

In short, neither `add` nor `main` know that the other function has variables with the same names. Because the scopes don't overlap, it's always clear to the compiler which `x` and `y` are being referred to at any time.

Key insight

Names used for function parameters or variables declared in a function body are only visible within the function that declares them. This means local variables within a function can be named without regard for the names of variables in other functions. This helps keep functions independent.

We'll talk more about local scope, and other kinds of scope, in a future chapter.

Where to define local variables

In modern C++, the best practice is that local variables inside the function body should be defined as close to their first use as reasonable:

```
1  #include <iostream>
2
3  int main()
4  {
5      std::cout << "Enter an integer: ";
6      int x{};       // x defined here
7      std::cin >> x; // and used here
8
9      std::cout << "Enter another integer: ";
10     int y{};       // y defined here
11     std::cin >> y; // and used here
12
13     int sum{ x + y }; // sum can be initialized with intended value
14     std::cout << "The sum is: " << sum << '\n';
15
16     return 0;
17 }
```

In the above example, each variable is defined just before it is first used. There's no need to be strict about this -- if you prefer to swap lines 5 and 6, that's fine.

Best practice

Define your local variables as close to their first use as reasonable.

As an aside...

Due to the limitations of older, more primitive compilers, the C language used to require all local variables be defined at the top of the function. The equivalent C++ program using that style would look like this:

```
#include <iostream>

int main()
{
    int x{}, y{}, sum{}; // how are these used?

    std::cout << "Enter an integer: ";
    std::cin >> x;

    std::cout << "Enter another integer: ";
    std::cin >> y;

    sum = x + y;
    std::cout << "The sum is: " << sum << '\n';

    return 0;
}
```

This style is suboptimal for several reasons:

- The intended use of these variables isn't apparent at the point of definition. You have to scan through the entire function to determine where and how each are used.
- The intended initialization value may not be available at the top of the function (e.g. we can't initialize `sum` to its intended value because we don't know the value of `x` and `y` yet).
- There may be many lines between a variable's initializer and its first use. If we don't remember what value it was initialized with, we will have to scroll back to the top of the function, which is distracting.

This restriction was lifted in the C99 language standard.

()Introduction to temporary objects (#temporaries)[3]

A **temporary object** (also sometimes called an **anonymous object**) is an unnamed object that is created by the compiler to store a value temporarily.

There are many different ways that temporary values can be created, but here's a common one:

```
#include <iostream>

int getValueFromUser()
{
    std::cout << "Enter an integer: ";
    int input{};
    std::cin >> input;

    return input; // return the value of input back to the caller
}

int main()
{
    std::cout << getValueFromUser() << '\n'; // where does the returned
value get stored?

    return 0;
}
```

In the above program, the function `getValueFromUser()` returns the value stored in local variable `input` back to the caller. Because `input` will be destroyed at the end of the function, the caller receives a copy of the value so that it has a value it can use even after `input` is destroyed.

But where is the value that is copied back to the caller stored? We haven't defined any variables in `main()`. The answer is that the return value is stored in a temporary object. This temporary object is then passed to `std::cout` to be printed.

Temporary objects have no scope at all (this makes sense, since scope is a property of an identifier, and temporary objects have no identifier).

Temporary objects are destroyed at the end of the full expression in which they are created. Thus the temporary object created to hold the return value of `getValueFromUser()` is destroyed after `std::cout << getValueFromUser() << '\n'` executes.

In the case where a temporary object is used to initialize a variable, the initialization happens before the destruction of the temporary.

In modern C++ (especially since C++17), the compiler has many tricks to avoid generating temporaries where previously it would have needed to. In the above example, since the return value of `getValueFromUser()` is immediately output, the compiler can skip creation and destruction of the temporary in `main()`, and use the return value of `getValueFromUser()` to directly initialize the parameter of `operator<<`.

Quiz time

Question #1

What does the following program print?

```cpp
#include <iostream>

void doIt(int x)
{
    int y{ 4 };
    std::cout << "doIt: x = " << x << " y = " << y << '\n';

    x = 3;
    std::cout << "doIt: x = " << x << " y = " << y << '\n';
}

int main()
{
    int x{ 1 };
    int y{ 2 };

    std::cout << "main: x = " << x << " y = " << y << '\n';

    doIt(x);

    std::cout << "main: x = " << x << " y = " << y << '\n';

    return 0;
}
```

Hide Solution (javascript:void(0))[4]

```
main: x = 1 y = 2
doIt: x = 1 y = 4
doIt: x = 3 y = 4
main: x = 1 y = 2
```

Here's what happens in this program:

- execution starts at the top of `main`
- `main`'s variable `x` is created and initialized with value `1`
- `main`'s variable `y` is created and initialized with value `2`
- `std::cout` prints `main: x = 1 y = 2`
- `doIt` is called with argument `1`
- `doIt`'s parameter `x` is created and initialized with value `1`
- `doIt`'s variable `y` is created and initialized with value `4`
- `doIt` prints `doIt: x = 1 y = 4`
- `doIt`'s variable `x` is assigned the new value `3`
- `std::cout` prints `doIt: x = 3 y = 4`
- `doIt`'s `y` and `x` are destroyed
- `std::cout` prints `main: x = 1 y = 2`
- `main` returns `0` to the operating system
- `main`'s `y` and `x` are destroyed

Note that even though `doIt`'s variables `x` and `y` had their values initialized or assigned to something different than `main`'s, `main`'s `x` and `y` were unaffected because they are different variables.

Next lesson
2.6 Why functions are useful, and how to use them effectively

5

Back to table of contents

6

Previous lesson
2.4 Introduction to function parameters and arguments

7

8

B U URL INLINE CODE C++ CODE BLOCK HELP!

Leave a comment...

Name*

Email*

Notify me about replies:

POST COMMENT

Find a mistake? Leave a comment above!

Avatars from https://gravatar.com/[10] are connected to your provided email address.

349 COMMENTS Newest

Divakar
April 7, 2024 10:11 am

Hello , Alex
You say variable destroy at the end of function doest it mean the value store in that variable in

RAM get delete

0 Reply

Alex Author
Reply to Divakar 11 April 8, 2024 4:32 pm

It means the memory allocated for the object is returned. The value stored in that memory location is typically left as-is, as it will likely be overwritten by the initializer of the next object to occupy that memory location.

1 Reply

IceFloe
February 20, 2024 12:54 am

1. can I intentionally control variables or hide their scope, or does this always happen automatically?
2. That is, if I define a local variable with the same identifier in different functions, then there will be no name conflict?
3. can a local variable jump to another function and be destroyed there (without copying the value to the parameters)?
4. when variables are destroyed, is a memory cell freed so that another local variable can occupy it?

1 Reply

Alex Author
Reply to IceFloe 12 February 22, 2024 1:28 pm

1. Not sure what you mean. Local variables are always scoped to the block in which they are defined. You can use this to your advantage.
2. Correct.
3. No.
4. Yes.

1 Reply

Mridul Thakur
February 6, 2024 9:43 pm

```cpp
#include <iostream>

int xx(){
    int y;
    return {y = 23};
    std::cout << y; // that line didn't run
}
int main(){
    std::cout << xx();
}
```

So, in a function objects get destroyed after return statement or after the end of a function?

Also, you said "The term “going out of scope” is typically applied to objects rather than identifiers."

But objects with a name are called variables, so they are also identifiers? What am i missing here?

0 Reply

Alex Author

Reply to Mridul Thakur 13 February 7, 2024 12:12 pm

At end of the function, which occurs after return statement executes or end of function is reached.

Variables are objects that have identifiers. Variables aren't identifiers, they have identifiers.

1 Reply

Divakar

Reply to Mridul Thakur 13 February 7, 2024 3:33 am

To print ' y' u have to initialized y first
2nd objects destroy at end of function that is after curly braces

0 Reply

Andy
January 24, 2024 2:38 am

We say an object goes out of scope at the end of the scope (the end curly brace) in which the object was instantiated. In the example above, the object named x goes out of scope at the end of the function main.

Did the object x go out of scope when the function doSomething() is run? We know that it is in scope before it, and not in scope during it. Or is that referring to its identifier?

0 Reply

Rosy
Reply to Andy [14] January 27, 2024 2:55 pm

Basically, local variables within a function are only accessible to statements within it's function. Therefore, think of this code:

```cpp
int doThing()
{
    int x{};
    return x;
}

int main()
{
    int y{};
    doThing();

    return 0;
}
```

x is in the scope of doThing(), and y is in the scope of main(). When we call doThing(), we lose access to y and gain access to x because we are now running code inside of doThing().

When doThing() finishes and we return to main(), we are running code inside of main() again so we can access y but no longer access x.

Last edited 2 months ago by Rosy

2 Reply

Andy
Reply to Rosy [15] January 27, 2024 3:10 pm

What you are saying conflicts with Alex's explanation below. When we call doThing(), you should not say that "y goes out of scope". It is true that y is no longer in scope any more, as it can no longer be used or accessed within doThing(), but it actually hasn't "gone out of scope". As Alex explained, which makes sense, the path of execution hasn't left the block that y was defined in, i.e. we are still in the "main block" if that makes sense, we just called a different function within the main block. The terminology sometimes overlap, but are referring to different things.

2 Reply

Rosy
Reply to Andy 16 January 27, 2024 3:39 pm

Thank you, I'm still getting used to the terms. I'll change my comment.

2 Reply

Alex Author
Reply to Andy 14 January 25, 2024 3:22 pm

No, variable `x` doesn't go out of scope when `doSomething` is run. It's still there, waiting for `doSomething` to return.

1 Reply

Andy
Reply to Alex 17 January 25, 2024 6:39 pm

But x cannot be seen or used within doSomething() right? So it is not in scope in doSomething(). Is this not considered x going out of scope? Or am I misunderstanding the terminology.

1 Reply

Alex Author
Reply to Andy 18 January 26, 2024 1:55 pm

> But x cannot be seen or used within doSomething() right?

Correct, `x` is not in scope at that point.

This is the confusion about use of the term "going out of scope". When we say "going out of scope", we're really talking about leaving the block the variable was defined in. So `x` hasn't "gone out of scope" (as the path of execution hasn't left the block `x` was defined in), but its "not in scope" either.

Last edited 2 months ago by Alex

1 Reply

Swaminathan
January 18, 2024 5:50 am

" In a program with lots of functions and function calls, variables are created and destroyed often."

THIS IS SPARTAA!!

1 Reply

Sam
January 2, 2024 10:53 pm

A local variable's lifetime ends at the point where it goes out of scope, so local variables are destroyed at this point.

Note that not all types of variables are destroyed when they go out of scope.

I feel these two statements are contradictory or not clear.
Are you trying to convey this - "Variable lifetime end ensures that variable goes out of scope. But a variable going out of scope DONOT mean variable reached its lifetime end". However for local variables going out of scope means variable reached its lifetime end.

1 Reply

Alex Author
Reply to Sam 19 January 5, 2024 5:46 pm

Scope defines where an identifier can be used. Lifetime defines when an object is created and destroyed.

> Variable lifetime end ensures that variable goes out of scope

No. Certain variables can have their lifetime ended before their identifiers go out of scope.

> But a variable going out of scope DONOT mean variable reached its lifetime end

Yes. Certain variables can go out of scope without the object reaching the end of its lifetime.

Put another way, scope and lifetime are independent properties.

> However for local variables going out of scope means variable reached its lifetime end.

Correct, for local variables there is a correlation between the two properties.

. For local variables, a variable goes out of scope at the same point its lifetime ends. There are other kinds of variables that are not d

2 Reply

Dedi
December 5, 2023 4:38 pm

```
#include <iostream>

int add(int x, int y) // add's x and y are created and enter scope here
{
// add's x and y are visible/usable within this function only
return x + y;
} // add's y and x go out of scope and are destroyed here

int x{ 5 };
int y{ 6 };

int main()
{
// main's x and y are usable within this function only
std::cout << add(x, y) << '\n'; // calls function add() with x=5 and y=6

return 0;
} // main's y and x go out of scope and are destroyed here
```

if i declare variables out side main function, is it going to be a problem someday?

0 Reply

Alex Author

Reply to Dedi [20] December 7, 2023 12:15 pm

Yes. Variables defined in the global scope are called global variables. You should avoid these for now.

0 Reply

theflash

November 6, 2023 8:42 pm

In the previous lesson, there was a line: "When the return statement is executed, the function exits immediately, and the return value is copied from the function back to the caller. This process is called return by value." But in this lesson, you said that
"Add copies of the value 11 back to caller main.
add parameters y and x are destroyed." So, how can parameters x and y be destroyed if a function is exited from the return statement?

1 Reply

Alex Author

Reply to theflash [21] November 10, 2023 11:55 am

When a function is exited (e.g. via a return statement, or via reaching the end of the function body), the function's stack frame is removed from the call stack and all local variables are destroyed. We discuss this a bit in lesson https://www.learncpp.com/cpp-tutorial/the-stack-and-the-heap/. You can read ahead if you're interested in the details.

0 Reply

Anon
September 21, 2023 8:03 am

good fun. tnx

Last edited 6 months ago by Anon

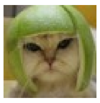
0 Reply

VexedBannister
September 19, 2023 11:57 pm

When I learned Java (and then C later), I was always taught to define variables at the top of a function. Personally I like it that way because then, when reading the rest of the code, I only have to worry about the actions being taken. It also means that I know all of the working parts before I start reading what the function does. It's like a primer for what's about to happen: You know that all these variables, with hopefully descriptive names, are going to be used at some point, and it gives you a framework with which to anticipate the code ahead. I often find that makes it easier for me to keep track of what is happening. Maybe it's just because I'm so used to it at this point.

Last edited 6 months ago by VexedBannister

0 Reply



Alex Author
Reply to VexedBannister 22 September 21, 2023 12:30 pm

In C++ it's not always possible to define everything up top (e.g. when it requires an initializer that isn't calculated until later in the function).

2 Reply

Mikey
Reply to Alex 23 December 31, 2023 10:38 pm

What's an example of an initializer being required? Why couldn't the variable be defined up top and then assigned later when its initializer is calculated?

0 Reply

Mikey
Reply to Mikey [24] January 2, 2024 1:25 pm

A constexpr variable must be immediately initialized (https://en.cppreference.com/w/cpp/language/constexpr#:~:text=it%20must%20be%20immediately%20initialized)[25]

0 Reply

Alex Author
Reply to Mikey [26] January 2, 2024 2:45 pm

A const variable must be initialized immediately. A constexpr variable is implicitly const, so it also qualifies.

Initialization only applies at the point where the object is instantiated.

If you were able to define a const or constexpr variable without an initializer, you wouldn't be able to assign it a value later because such variables can't be modified after initialization.

0 Reply

Aditya Raj
August 11, 2023 10:09 pm

I'm confused regarding the initialization of parameters. In previous section (2.4), it was said that "Function parameters are initialized by value provided by caller of the function." But here it says "Function parameters are created and initialized when function is entered". Aren't these two statements contradictory?

Also about the latter statement, initialization means creating and providing a value to the variable there only right? But there is no value provided to parameters when a function is entered, it is only done during function call. Sorry if I'm missing something obvious here, I'd really appreciate if someone could clear this doubt.

0 Reply

Alex Author
Reply to Aditya Raj [27] August 12, 2023 5:22 pm

Not contradictory. "entered" means when the function is called.

So, when a function is called, the caller provides arguments to the function. The parameters of the function are instantiated and initialized with these arguments. The function executes. At the end of the function, the parameters are destroyed, and control returns to the caller.

1 Reply

Aditya Raj
Reply to Alex [28] August 12, 2023 8:33 pm

Ah that makes sense, thank you!

0 Reply

N T
August 4, 2023 5:20 pm

> The above rules around creation, initialization, and destruction are guarantees. That is, objects must be created and initialized no later than the point of definition

Just to clarify, does the initialisation part of this guarantee only apply if an initialiser has been provided at the point of definition? So, "objects must be created and initialised (if an initialiser value is given) no later than the point of definition"?

1 Reply

Alex Author
Reply to N T [29] August 6, 2023 7:45 pm

> does the initialisation part of this guarantee only apply if an initialiser has been provided at the point of definition?

The variable will be initialized by the point of definition if any kind of initialization value exists.

The initialization value does not necessarily have to be provided at the point of definition. There are some cases where initialization values can be specified elsewhere, or where a default initialization value is applied if none is explicitly specified. These still must be applied no later than the point of definition, otherwise the object could be usable prior to being initialized.

2 Reply

Krishnakumar
Reply to Alex [30] August 24, 2023 12:59 am

>otherwise the object **could be usable** prior to being initialized.

Typo?

0 Reply

oldjak
Reply to Krishnakumar [31] August 31, 2023 12:56 pm

not a typo

0 Reply

Krishnakumar
Reply to oldjak [32] October 10, 2023 6:32 am

Ah indeed. `usable` in American English, vs `useable` in British English.

1 Reply

Kasaquiel
July 7, 2023 5:28 pm

Great lesson, but I have a question about the quiz, shouldnt there be a return statement in the doIt function since it is not a main function?

Last edited 9 months ago by Kasaquiel

1 Reply

Alex Author
Reply to Kasaquiel [33] July 9, 2023 5:11 pm

Return statements are only required for value-returning functions (those that have a return type other than `void`).

4 Reply

samuel
June 9, 2023 8:15 am

Essentially each lesson requires me 1h 30m aprox for fully understanding. Amazing course. Made me even pull a paper out and start taking notes

5 Reply

taotao
Reply to samuel [34] December 2, 2023 2:41 am

i think it is a good method.

1 Reply

Ivanovich
April 27, 2023 8:33 am

I don't get it why the doIt() function is called twice? And also how come the second time it's called x is now 3?

0 Reply

Andrew
Reply to Ivanovich 35 July 26, 2023 8:38 am

I find it alot easier to figure out what functions do by adding them to my ide (without compiling) and making notes to what they do

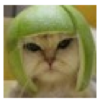
```cpp
#include <iostream>

void doIt(int x)
{
    // y is created with value 4
    int y{ 4 };
    std::cout << "doIt: x = " << x << " y = " << y << '\n';
    // x is assigned the value 3
    x = 3;
    std::cout << "doIt: x = " << x << " y = " << y << '\n';
}

int main()
{
    int x{ 1 };
    int y{ 2 };
    // outputs 1, 2 because main is setting x to 1, and y to 2.
    std::cout << "main: x = " << x << " y = " << y << '\n';
    // will output 1, 4 because doit calls for x and y to be
printed before x is assigned a variable
    doIt(x);
    // ^will output 3, 4 because now x is assigned a variable and
is asking again to produce x and y


    // will produce 1, and 2 because its using the variables in
local scope of int main
    std::cout << "main: x = " << x << " y = " << y << '\n';
    return 0;
}
```

in this note before i compiled i was actually incorrect, going back you can see that when y is changed to 4 its outputting x and y but since the value x in int mains local scope is set to 1 then it is 1,4 since x doesnt have time to get updated.

then when x is updated it asks for x and y again which at that point x has been changed to 3.

going back to the final output, the local scope x and y from main() are being used so they will result in x being 1 and y being 2 again

0 Reply



Rescued

Reply to Ivanovich 35 April 28, 2023 1:50 am

Hello! Let me try to clarify your confusion

doIt() Function is not called twice. it's called only one. upon the function being called by the caller main(). the compiler leaves a bookmark and jumps to the function that is called which is doIt()

```
void doIt(int x)
{
int y{ 4 };
std::cout << "doIt: x = " << x << " y = " << y << '\n';

x = 3;
std::cout << "doIt: x = " << x << " y = " << y << '\n';
}
```

This is the function doIt(). if you notice. the first 2 lines. variable int y is being defined and initialized with the value 4.
Then the next line is std::cout << "doIt: x = " << x << " y = " << y << '\n'; the result of it is "doIt X=1 Y=4"
we got value 1 for variable x from the argument that already exists in the main() so we used it for x in this case.
and we got the value 4 for variable y from the variable that is defined inside the function doIt()
so it turn out to be "doIt X=1 Y=4"

Now.. the function is not called twice, instead, if you look carefully inside the function body, "doIt: x = " << x << " y = " << y << '\n'; WAS written again. This time the result turned out to be doIt x=3 y=4 Unlike the first time, the X value has changed.

Why is that? well. the first time, we used the argument that in main() for the variable x that is inside function doIt()
The second time. we used the value of x = 3; Instead of using the arguments.
The first time x =3; was OUT of scope. "So the compiler didn't know it exist. so he used the argument in main()"
The second time x=3; was IN THE SCOPE. "So the compiler used it and didn't use the argument in main()"

Last edited 11 months ago by Rescued

2 Reply

thibault

April 22, 2023 12:19 pm

Hi, i'm a bit confused about the fact to define variable closer to their use.
In my school (school 42) we have a norm who force us to define our variables at the top of the function body. Isn't it a better practice to declare at the top to have a clear view of the variables in the scope ?
Thank you, amazing course by the way :)

0 Reply

Alex Author
Reply to thibault [36] April 24, 2023 12:13 am

No. And why do you need a clear view of the variables in the scope?

It's better to define them only as needed. The context in which they are defined can be important to understanding how they are used, and we are more likely to have a value we can initialize them with at that point.

Define at the top is a style that has long since fallen out of favor.

4 Reply

thibault
Reply to Alex [37] April 24, 2023 5:29 am

You right, I read a bit about it. Until now, we're only developing using C and declare variable at the top is more common in C as I read (historical context). I'll now declaring it close to their first use, will be weird at first because I'm use to declare at the top but need to become use with that. Thank you

2 Reply

Timo
April 21, 2023 2:35 pm

int x; is a definition? This is a declaration right?

0 Reply

Alex Author
Reply to Timo [38] April 23, 2023 4:59 pm

It's both.

It's a declaration because it introduces a name and type to the compiler. It's a definition because it causes an object to be instantiated.

0 Reply

Timo
Reply to Alex 39 April 23, 2023 11:57 pm

Ok that clears things up!

0 Reply

DeadlyAhmed
April 20, 2023 3:41 pm

I still can't get what's really done "pass by value", when a function is called by a caller, the passed arguments by the caller do which:

1. Assign their values to the parameters (copy) means that there are already variables there (parameters)
2. they don't assign(no variables at all in the function definition), they CREATE new variables (old parameters) and initialize them

0 Reply

I M
Reply to DeadlyAhmed 40 April 22, 2023 3:15 am

it's the second one, when you call a function it copies the parameters you pass to it and it makes them a new variable. this is so that you can alter the variables without changing the original version of them.

0 Reply

DeadlyAhmed
Reply to DeadlyAhmed 40 April 20, 2023 3:43 pm

I go with 2nd one but I'm not sure if is this right or wrong.

0 Reply

Ahmadi Badreddine
April 1, 2023 10:58 pm

what about global variables ?

0 Reply

Alex Author
Reply to Ahmadi Badreddine 41 April 2, 2023 8:11 pm

Those are covered in chapter 6.

1 Reply

Vishal
March 23, 2023 5:01 am

Why is the term variable and object used interchangeably? Is there any specific difference between them?

1 Reply

Alex Author
Reply to Vishal 42 March 24, 2023 7:29 pm

A variable is an object that has been given a name. So when we talk about objects, we generally mean variables too (since they are objects).

The distinction is made because C++ also supports anonymous / temporary objects that do not have a name. And in some cases, objects with names and objects without names behave differently.

3 Reply

Emeka Daniel
March 17, 2023 5:31 am

Since lifetime is a runtime property and scope is a compiletime property. Wouldn't it be better to refer to it as an object(a region of storage that is assigned a memory address) is attributed a lifetime(point of instantiation to destruction), while an identifier(variable name) is attributed a scope(defines accessibility within a source code).

This way, I think is more easier to differentiate, because they seem so similar that one might confuse the two. Just saying..

0 Reply

Alex Author
Reply to Emeka Daniel March 18, 2023 1:37 pm

I think the article already does that when defining the terms. From the article:

- "an object's lifetime is defined to be the time between its creation and destruction"
- "An identifier's scope determines where the identifier can be seen and used within the source code."

1 Reply

Emeka Daniel
Reply to Alex March 19, 2023 1:14 pm

Oh, yh

Thanks for the reply though

1 Reply

YouAreAwesome
March 14, 2023 4:55 am

Many things are opinions, but to me, you've made this so easy and clear to understand that it's really amazing! Thanks so much for your knowledge and efforts!

1 Reply

Krishnakumar
January 22, 2023 9:22 pm

For the function `add()`, since C++ does not guarantee that function parameters are to be evaluated from left to right, do we need a note saying that the comments correspinding to the header line and last curly braces line about lifetime and scope are not necessarily guaranteed to be in that order?

1 Reply

Alex Author
Reply to Krishnakumar January 25, 2023 10:15 am

Feels a little early to me to be introducing the concept of unspecified evaluation order. That's brought up later in this chapter. In the detailed explanation bullet-point list, I collapsed the lines discussing the creation of x and y into a single line so it no longer implies that `y` is created after `x`.

3 Reply

Krishnakumar
Reply to Alex May 25, 2023 4:27 pm

I agree that this is fine for now, to avoid confusion. Nevertheless, I just wish to draw your attention that more recent comments from you in the past couple of lessons have already mentioned unspecified evaluation order.

It is somewhat notable (stands-out) that the word ordering of `y` and `x` getting created in the function's annotation is in the reverse order to the corresponding annotation next to the calling line of code.

It's a tough one pedagogically, but probably fine as it is now.

0 Reply

Boris Pulatov

Reply to Krishnakumar January 25, 2023 6:04 am

I'm with Krishnakumar on this. Superficially, the compiler is free to do the following two in either order:

- add's variable x is created and initialized with value 5
- add's variable y is created and initialized with value 6

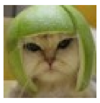
0 Reply

badhat

January 7, 2023 4:07 pm

>>> add copies the value 11 back to caller main
You lost me there, how can `add` copy over `11` to main then end the lifetime of its local variables after? Wouldn't that mean that `x` and `y` are destroyed in the caller, main, instead of the end of the curly brace for `add`?

0 Reply



Alex Author

Reply to badhat January 7, 2023 4:30 pm

Assuming no optimizations, a temporary object is created in the scope of the caller, which is initialized with the return value. When the called function ends, the local variables are destroyed, and control returns to the caller. The temporary object (containing the return value) still exists at that point since it is in the scope of the caller, not the called function.

1 Reply

Krishnakumar

Reply to Alex October 10, 2023 6:34 am

Fascinating. Perhaps this detail ought to be present in a suitable later lesson?

0 Reply

randomGuy

December 18, 2022 12:23 pm

Instead of:

```
#include <iostream>

int main()
{
std::cout << "Enter an integer: ";
int x{}; // x defined here
std::cin >> x; // and used here

std::cout << "Enter another integer: ";
int y{}; // y defined here
std::cin >> y; // and used here

int sum{ x + y }; // sum defined here
std::cout << "The sum is: " << sum << '\n'; // and used here

return 0;
}
```

As you suggested, this one is redundant so I prefer what I have written below.

```
#include <iostream>

int sumNumbers()

{
std::cout << "Enter an integer: ";
int x{};
std::cin >> x;
return x;

}

int main()

{
int x{ sumNumbers () };
int y{ sumNumbers () };
std::cout << x << " + " << y << " = " << x + y << '\n';
return 0;

}
```

I understand concepts for now, but sometimes I am confused with syntax when I make my programs. Is it normal? I started two days ago

0 Reply

lizardking

Reply to randomGuy December 27, 2022 10:24 am

Yes being confused with syntax is normal. When you first learn a written language you don't remember all the grammar rules right away. Even veteran programmers will often have to look up syntax specifics or make corrections after a program fails its first compile.

Secondly, make sure your function names make sense. You created a function called "sumNumbers" but is that function actually used to sum numbers? No, it is just collecting input from the user. The addition is being done later in main. A better name for that function would be something like "getValueFromUser()," which was created in chapter 2.3.

Last edited 1 year ago by lizardking

3 Reply

hantao
October 29, 2022 11:38 pm

Excellent tutorial. I am still confused about "identifiers in/out of scope" vs "objects going out of scope" and "runtime property" vs "complie_time property".

How will the differences affect our program? Or they are just terminology?

0 Reply

Alex Author
Reply to hantao October 31, 2022 11:24 am

They are just terminology.

A compile-time property is one that is known/used at compile-time. A runtime property is one that is known/used when the program is running.

2 Reply

Question on run-time
September 4, 2022 1:20 pm

>>we discussed how a variable definition such as int x; causes the variable to be instantiated (created) when this statement is executed. Function parameters are created and initialized when the function is entered, and variables within the function body are created and initialized at the point of definition.

Does it mean all the variable creation and initialization happening at run-time either they are defined as function parameters or defined inside a function body?

0 Reply

Alex Author
Reply to Question on run-time September 13, 2022 3:00 pm

No. Later in this series, we discuss global variables, which are defined outside a function body, but can still be initialized at runtime (when the program starts).

3 Reply

Austin
August 13, 2022 6:38 pm

So in the "Another Example" in the “Out of scope” vs “going out of scope”, is a and be entirely different than x and y? Does the return x+y do anything in that situation? Or does x and y somehow magically become a and b, and it just works?

0 Reply

Alex Author
Reply to Austin August 15, 2022 11:30 am

Yes, `a` and `b` are separate variables than `x` and `y`. When function `add(x, y)` is called, variable `a` is created and initialized with the value of `x` and `b` is created and initialized with the value of `y`. This doesn't affect `x` and `y`.

The return `x+y` just returns a value back to the caller, which it uses to print. It's not particularly relevant to the example.

1 Reply

Paul
July 21, 2022 10:13 am

#include <iostream>

int doAddition(int x, int y) //Does addition
{
int sum{};
sum = x + y;

return sum;
}

int main()
{
std::cout << "enter an integer ";
int num1;
std::cin >> num1;
std::cout << "enter another integer ";
int num2;
std::cin >> num2;

std::cout << doAddition(num1, num2) << '\n';

return 0;
}

//First Attempt, any thoughts?

0 Reply

tenchu
Reply to Paul September 1, 2022 10:32 am

Nothing really wrong with your code, good job!
But there is room for improvement, no need to make a variable for something you will just return.

int doAddition(int x, int y)
{
return x + y;
}

Also remember DRY(Do not repeat yourself)? You could make a function to get the input, of course you can remove the std::cout outside the function if you really want "enter another integer". the "input" variable only exists inside this function, so it doesnt matter it has "the same name".

int getValue()
{
std::cout << "Enter a interger: ";
int input{};
std::cin >> input;

return input;
}

Then you can call it in main like this:
int main()
{
std::cout << doAddition(getValue(), getValue());

return 0;
}

1 Reply

Vince
May 19, 2022 6:04 pm

Thanks for the tutorial!

I just noticed in this sentence under the **Local Scope** section: "A local variable's scope begins at the point of variable definition, and stops at the end of the set of curly braces in which they are defined (or for function parameters, at the end of the function)", should the part "in which they are defined" be "in which it is defined", since you're talking about a variable, instead of variables?

0 Reply

Links

1. https://www.learncpp.com/author/Alex/
2. https://www.learncpp.com/cpp-tutorial/introduction-to-objects-and-variables/
3. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#temporaries
4. javascript:void(0)
5. https://www.learncpp.com/cpp-tutorial/why-functions-are-useful-and-how-to-use-them-effectively/
6. https://www.learncpp.com/
7. https://www.learncpp.com/cpp-tutorial/introduction-to-function-parameters-and-arguments/
8. https://www.learncpp.com/introduction-to-local-scope/
9. https://www.learncpp.com/cpp-tutorial/void/
10. https://gravatar.com/
11. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-595509
12. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-593849
13. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-593347
14. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-592795
15. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-592966
16. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-592967
17. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-592882
18. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-592894
19. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-591702
20. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-590578
21. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-589508
22. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-587454
23. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-587549
24. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-591573
25. https://en.cppreference.com/w/cpp/language/constexpr#:~:text=it%20must%20be%20immediately
26. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-591659
27. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-585424
28. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-585510
29. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-585096
30. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-585193

31. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-586039
32. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-586448
33. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-583395
34. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-581500
35. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-579866
36. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-579656
37. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-579749
38. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-579600
39. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-579714
40. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-579556
41. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-578863
42. https://www.learncpp.com/cpp-tutorial/introduction-to-local-scope/#comment-578612